Intel® Distribution for Python* Scaling HPC and Big Data

Sergey Maidanov

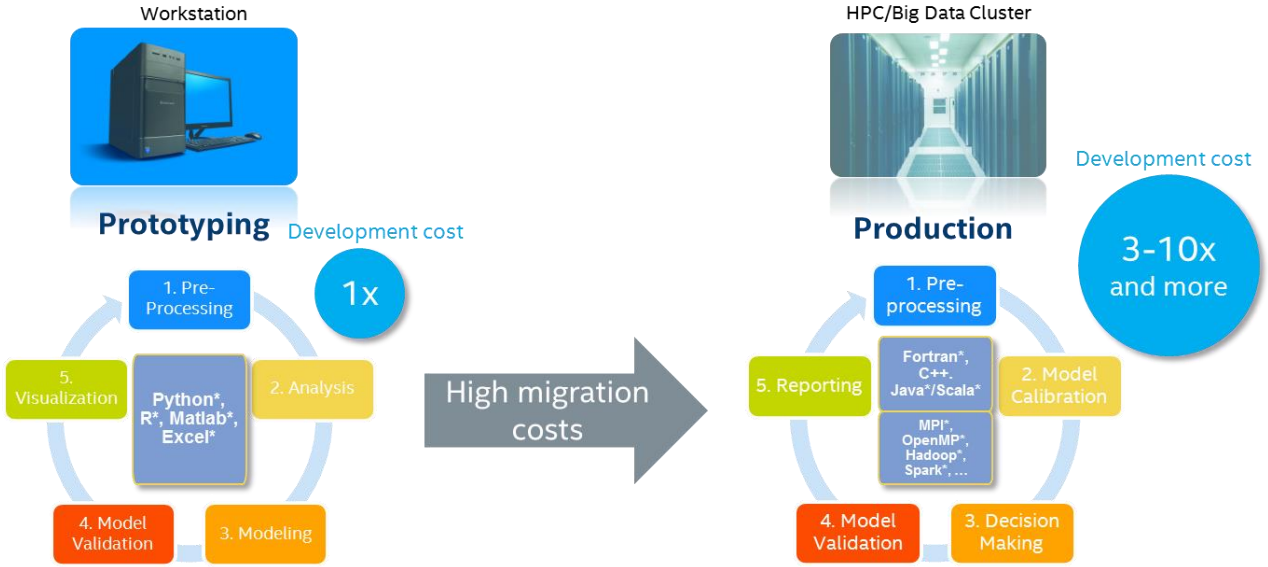
Software Engineering Manager for

Intel® Distribution for Python*

Quick Facts

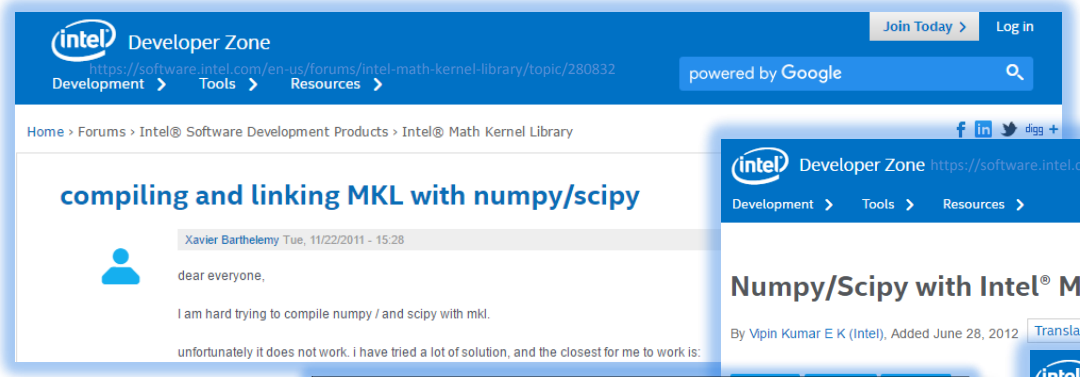
- Released Intel® Distribution for Python* in Sep'16
  - Out-of-the-box experience in HPC and Data Science, pip and conda support
  - Near-native performance for Linear Algebra, initial optimizations in FFT and NumExpr
  - Introduced TBB for threading composability, random_intel for fast RNG, pyDAAL
- Update release in Oct'16
  - Greater compatibility with Anaconda*
  - Performance and usability enhancements
  - Neural networks support in pyDAAL
  - Docker images
- Update 2 release in Feb'17
  - Memory optimizations in NumPy
  - Umath optimizations in NumPy
  - NumPy and SciPy FFT improvements
  - Scikit-learn optimizations
  - New pyDAAL features

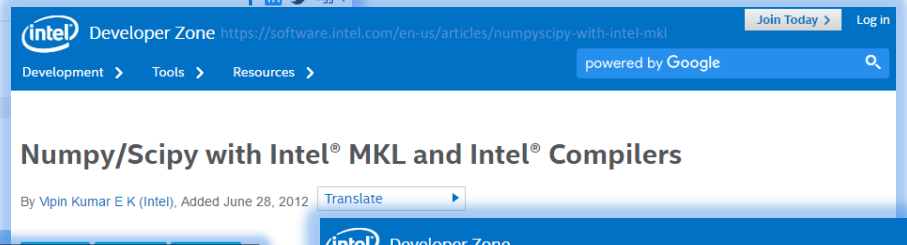
What Problems We Solve: Scalable Performance

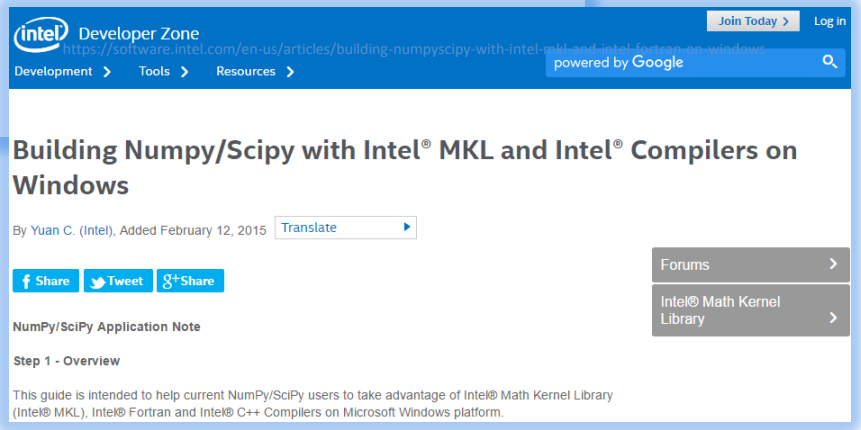


Make Python usable beyond prototyping environment by scaling out to HPC and Big Data environments

What Problems We Solve: Ease of Use







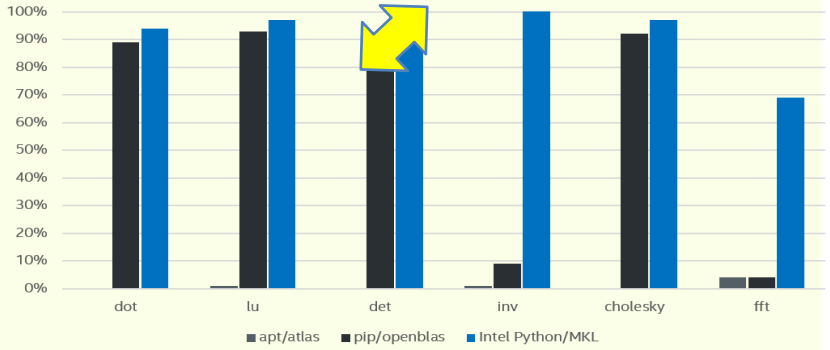
"Any articles I found on your site that related to actually using the MKL for compiling something were overly technical. ***I couldn't figure out what the heck some of the things were doing or talking about.*** *"* – Intel® Parallel Studio 2015 Beta Survey Response

Why Yet Another Python Distribution?

Mature AVX2 instructions based product

Intel® Xeon® Processors

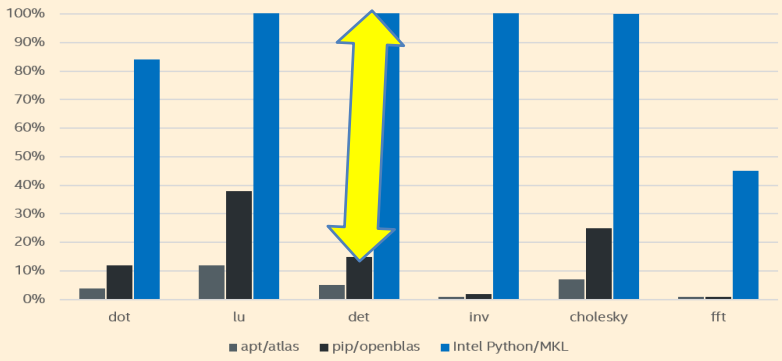
Python* Performance as a Percentage of C/Intel® MKL for Intel® Xeon® Processors, 32 Core (Higher is Better)



New AVX512 instructions based product

Intel® Xeon Phi™ Product Family

Python* Performance as a Percentage of C/Intel® MKL for Intel® Xeon Phi™ Product Family, 64 Core (Higher is Better)



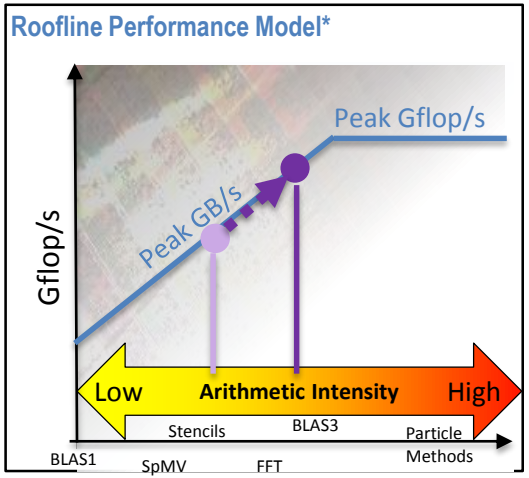
Configuration Info: apt/atlas: installed with apt-get, Ubuntu 16.10, python 3.5.2, numpy 1.11.0, scipy 0.17.0; pip/openblas: installed with pip, Ubuntu 16.10, python 3.5.2, numpy 1.11.1, scipy 0.18.0; Intel Python: Intel Distribution for Python 2017;. Hardware: Xeon: Intel Xeon CPU E5-2698 v3 @ 2.30 GHz (2 sockets, 16 cores each, HT=off), 64 GB of RAM, 8 DIMMS of 8GB@2133MHz; Xeon Phi: Intel Intel® Xeon Phi™ CPU 7210 1.30 GHz, 96 GB of RAM, 6 DIMMS of 16GB@1200MHz

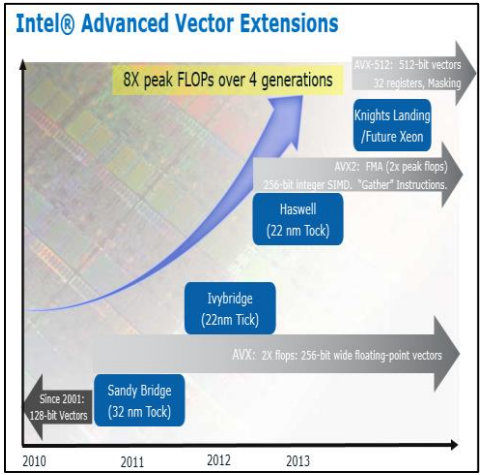
Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. * Other brands and names are the property of their respective owners. Benchmark Source: Intel Corporation

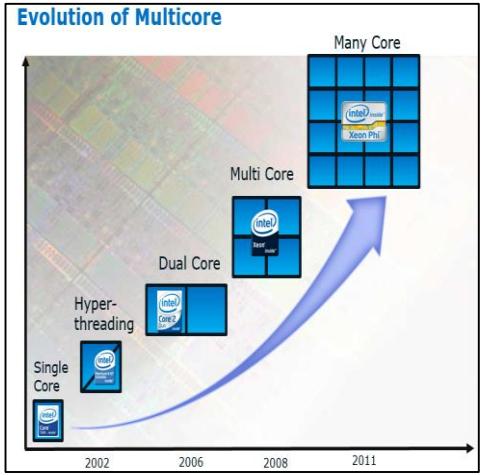
Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804 .

Scaling To HPC & Big Data Environments

- Hardware and software efficiency crucial in production (Perf/Watt, etc.)
- Efficiency = Parallelism
  - Instruction Level Parallelism with effective memory access patterns
  - SIMD
  - Multi-threading
  - Multi-node





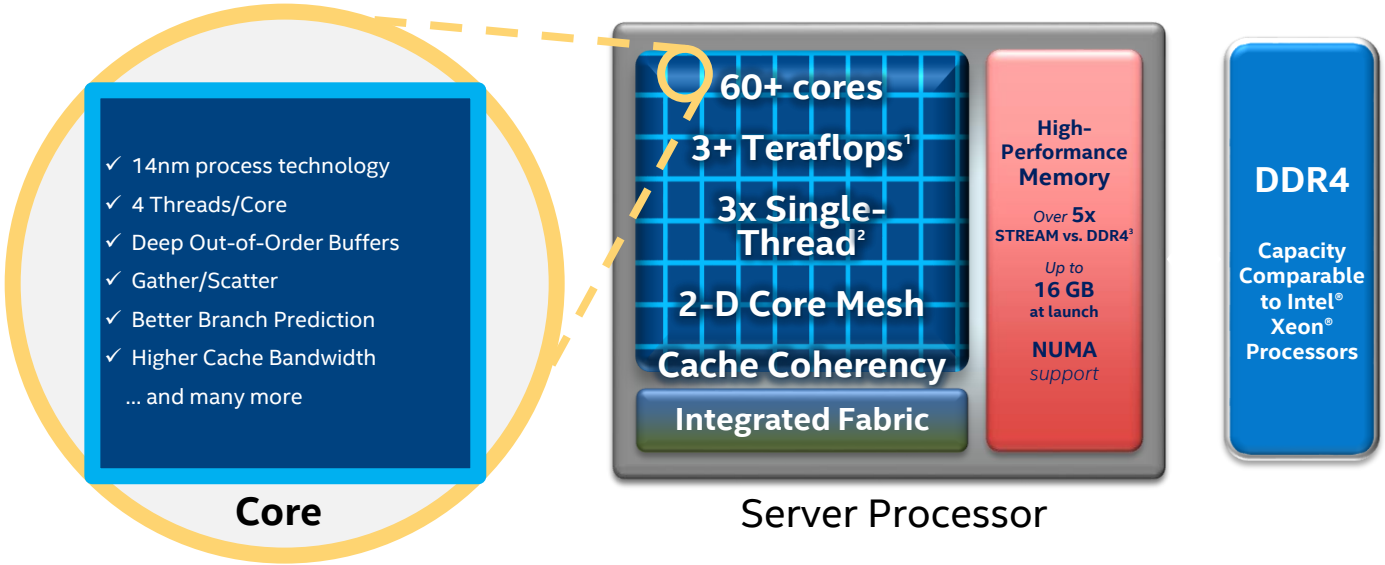


* Roofline Performance Model https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/

Intel® Xeon Phi™ x200 (Knights Landing) processor

Architectural Enhancements = ManyX Performance

Based on Intel® Atom™ core (based on Silvermont microarchitecture) with Enhancements for HPC



From SSE or AVX to AVX-512: Setting Right Expectations

- **2x vector length is typically <2x performance boost**
  - Applications have scalar sections, so are subject to Amdahl's Law
  - Some applications are limited by access to data
    - If throughput bound, MCDRAM may help
    - If latency bound, prefetching may help
  - Loops may need larger trip counts to get full benefit
- **Gains from newly vectorized loops can be large**
- **Application hotspots may change significantly between AVX and AVX512 codes**

Efficiency = Parallelism

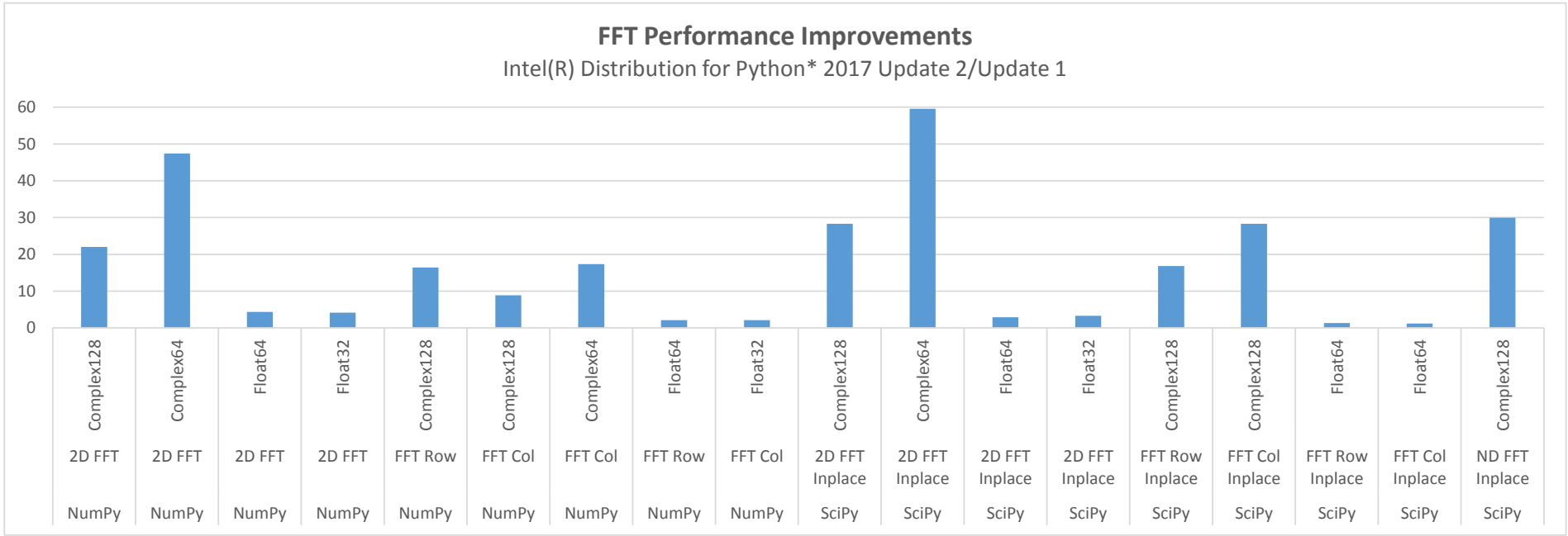
- CPython as interpreter inhibits parallelism but…
- … Overall Python tools evolved far toward unlocking parallelism

Packages (numpy*, scipy*, scikit-learn*, etc.) accelerated with MKL, DAAL, IPP	Composable multi-threading with Intel® TBB and Dask*	Multi-node parallelism with mpi4py* accelerated with Intel® MPI*
Language extensions for vectorization & multi-threading (Cython*, Numba*)	Integration with Big Data platforms and Machine Learning frameworks (pySpark*, Theano*, TensorFlow*, etc.)	Mixed language profiling with Intel® VTune™ Amplifier

Out-of-the-box performance with accelerated numerical packages

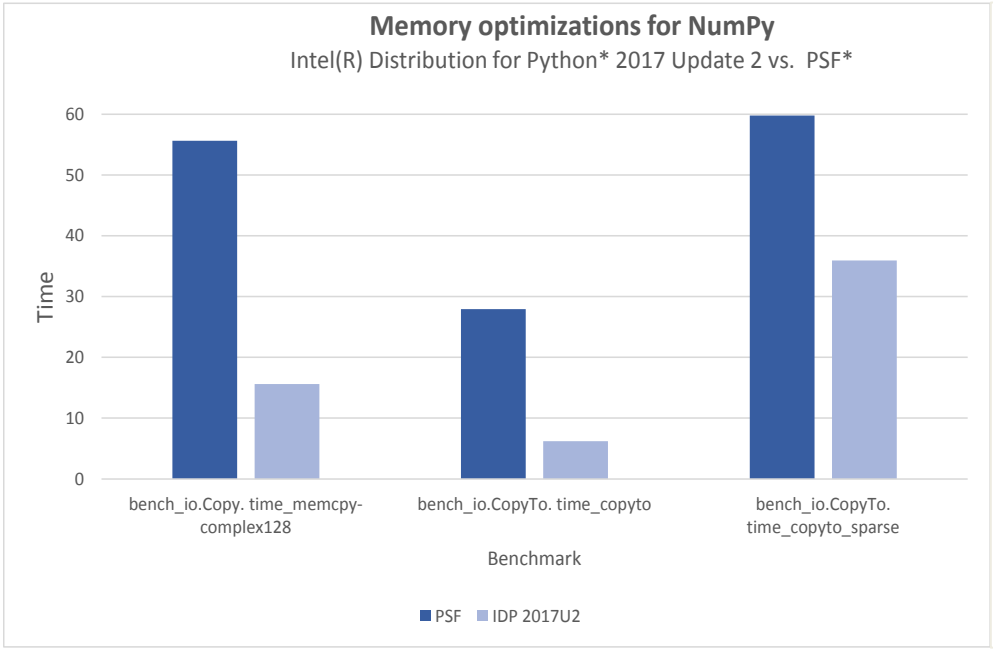
Widespread optimizations in NumPy & SciPy FFT

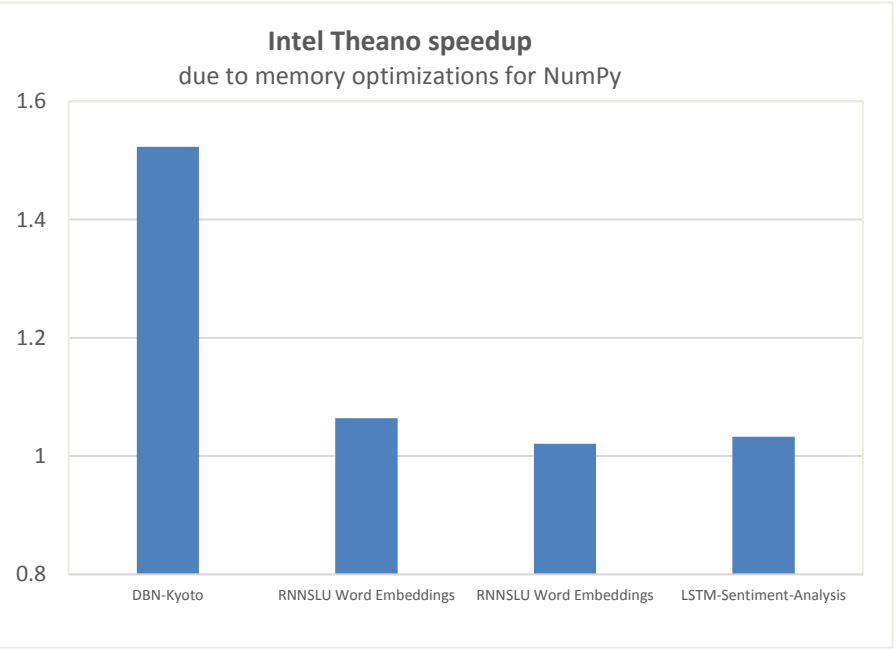
- Up to 60x improvement in FFT for the range of different use cases in NumPy and SciPy



Memory optimizations for NumPy arrays

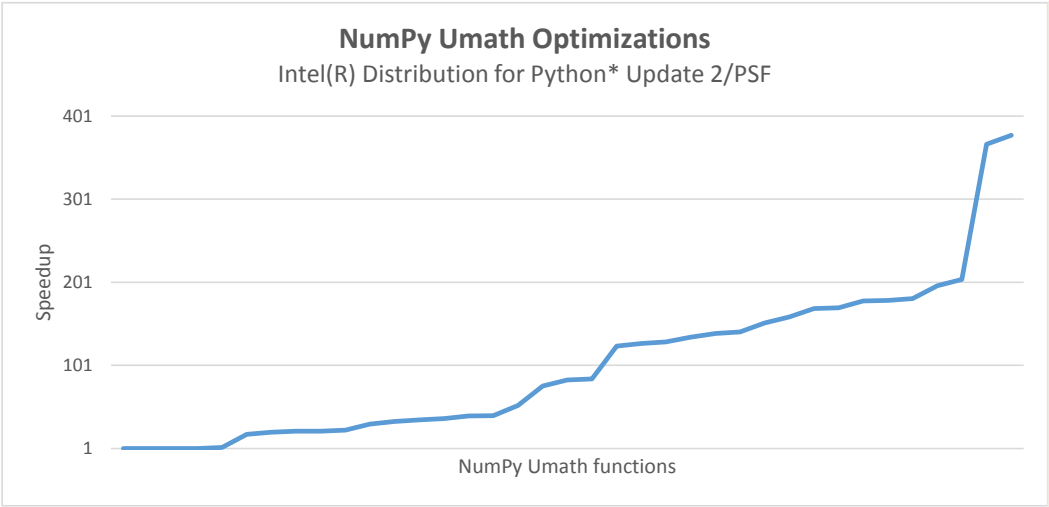
- Optimized array allocation/reallocation, copy/move
  - Memory alignment and data copy vectorization & threading

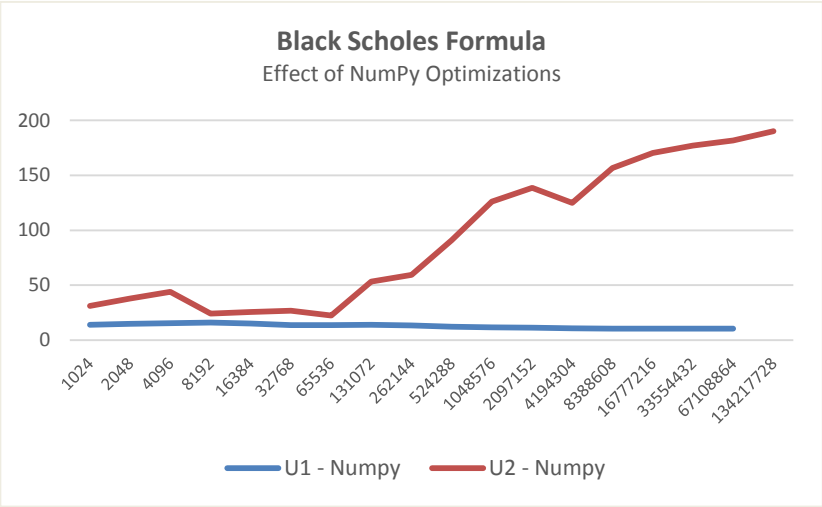




Optimizations for NumPy umath functions

- Optimized arithmetic/transcendental expressions on NumPy arrays
  - Up to 400x better performance due to vectorization & threading
  - 180x speedup for Black Scholes formula due to umath optimizations





Choosing right alternative for the best parallelism

Benchmark: Black Scholes Formula

- Problem: Evaluate fair European call- and put-option price, V_{call} and V_{put}, for underlying stock
- Model Parameters:
  - S_0 – present underlying stock price
  - X – strike price
  - σ - stock volatility
  - r – risk-free rate
  - T - maturity
- In practice one needs to evaluate many (*nopt*) options for different parameters

$$V_{call} = S_0 \cdot \mathrm{CDF}(d_1) - e^{-rT} \cdot X \cdot \mathrm{CDF}(d_2)$$

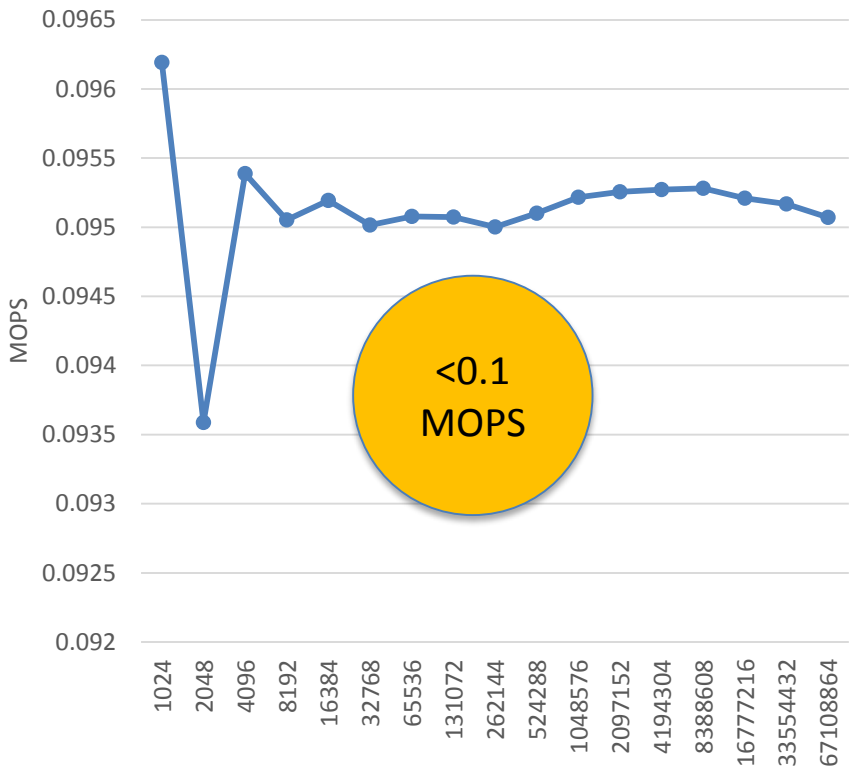
$$V_{put} = e^{-rT} \cdot X \cdot \mathrm{CDF}(-d_2) - S_0 \cdot \mathrm{CDF}(-d_1)$$

$$d_1 = \frac{\ln\left(S_0/X\right) + \left(r + \sigma^2/2\right)T}{\sigma\sqrt{T}}$$

$$d_2 = \frac{\ln\left(S_0/X\right) + \left(r - \sigma^2/2\right)T}{\sigma\sqrt{T}}$$

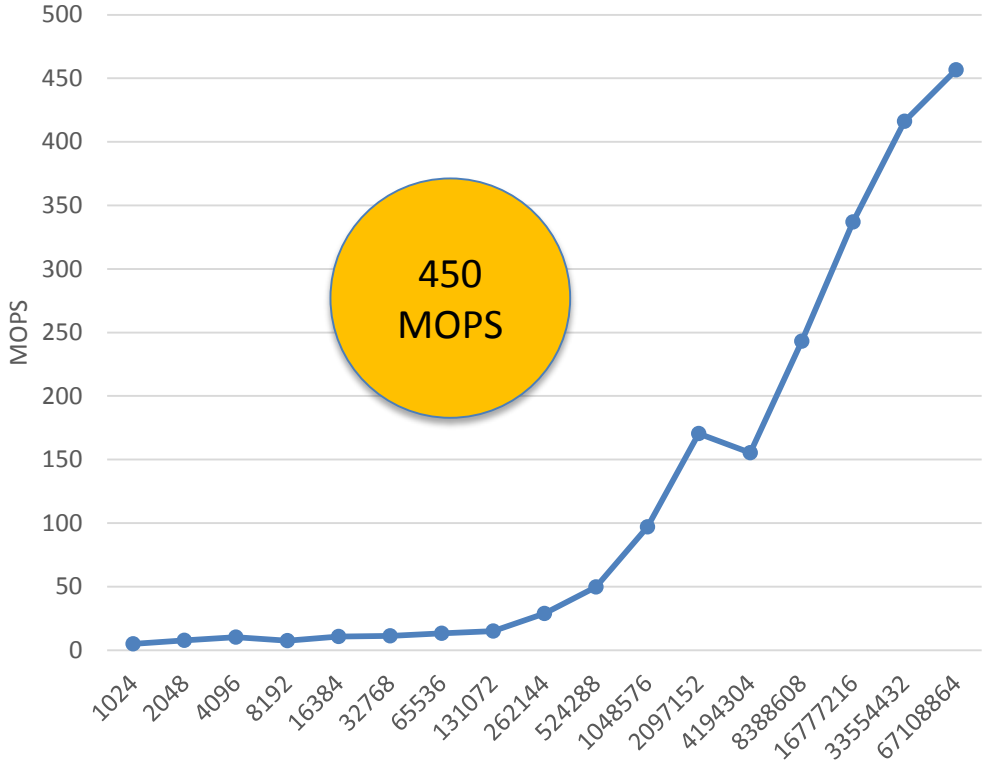
Good performance benchmark for stressing VPU and memory

Variant 1: Plain Python



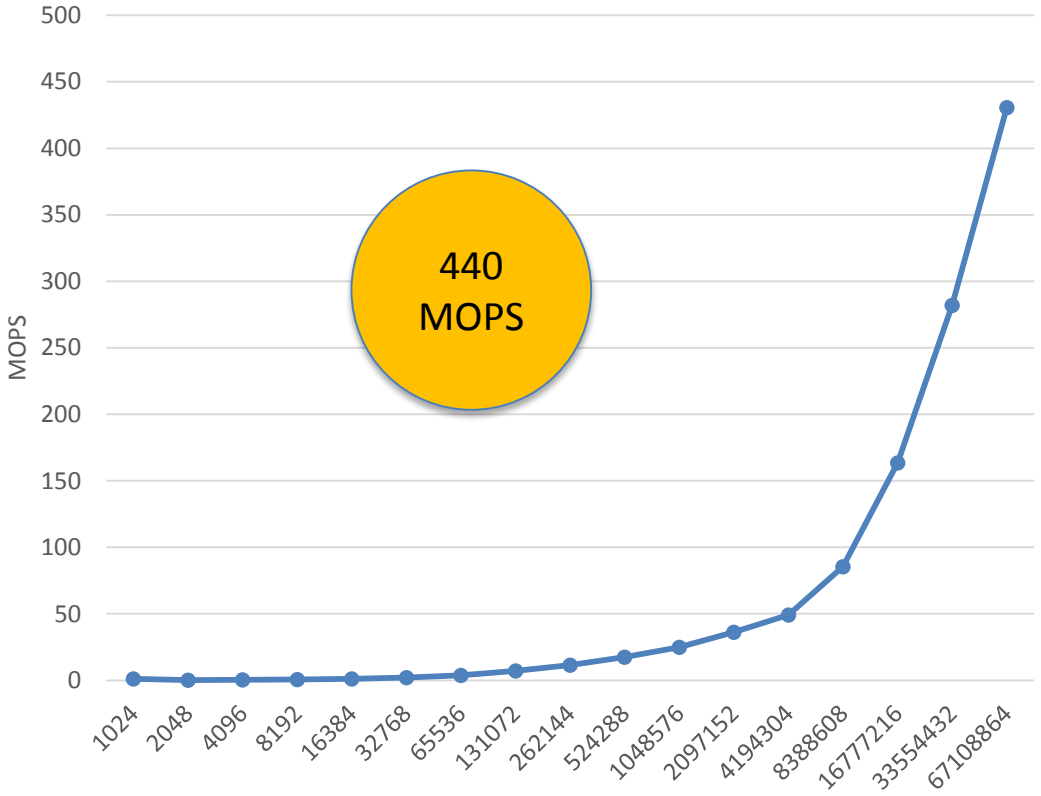

```
6  def black_scholes ( nopt, price, strike, t, rate, vol, call, put ):
7      mr = -rate
8      sig_sig_two = vol * vol * 2
9
10     for i in range(nopt):
11         P = float( price [i] )
12         S = strike [i]
13         T = t [i]
14
15         a = log(P / S)
16         b = T * mr
17
18         z = T * sig_sig_two
19         c = 0.25 * z
20         y = 1/sqrt(z)
21
22         w1 = (a - b + c) * y
23         w2 = (a - b - c) * y
24
25         d1 = 0.5 + 0.5 * erf(w1)
26         d2 = 0.5 + 0.5 * erf(w2)
27
28         Se = exp(b) * S
29
30         call [i] = P * d1 - Se * d2
31         put [i] = call [i] - P + Se
```

Variant 2: NumPy* arrays and Umath functions



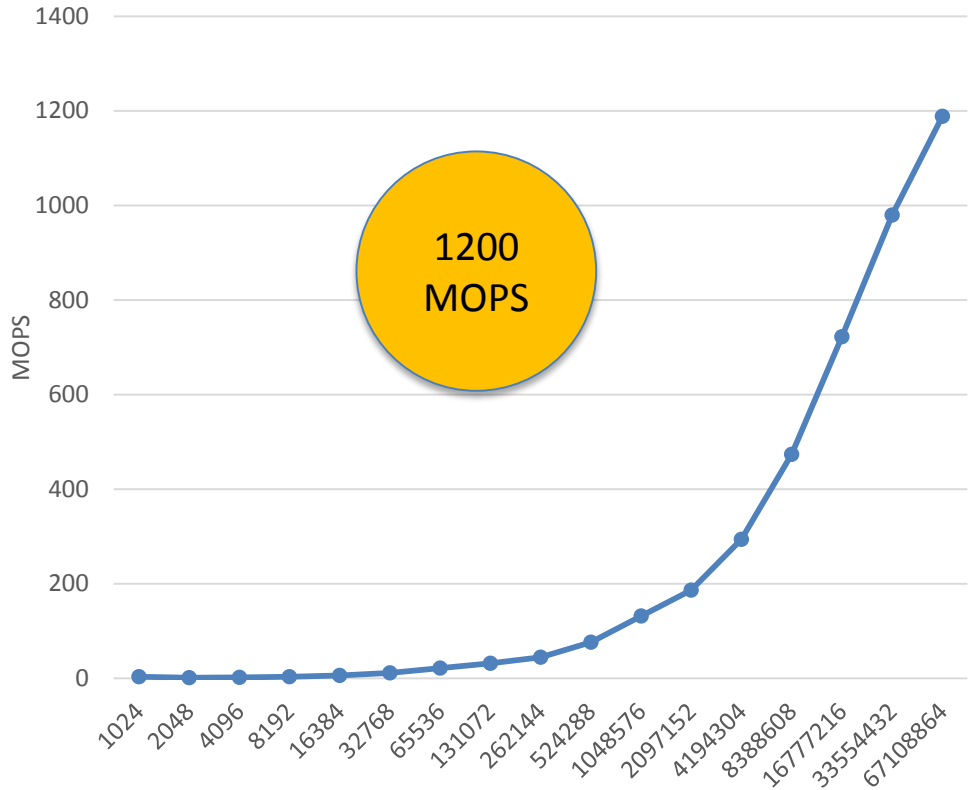

```python
def black_scholes ( nopt, price, strike, t, rate, vol ):
    mr = -rate
    sig_sig_two = vol * vol * 2

    P = price
    S = strike
    T = t

    a = log(P / S)
    b = T * mr

    z = T * sig_sig_two
    c = 0.25 * z
    y = invsqrt(z)

    w1 = (a - b + c) * y
    w2 = (a - b - c) * y

    d1 = 0.5 + 0.5 * erf(w1)
    d2 = 0.5 + 0.5 * erf(w2)

    Se = exp(b) * S

    call = P * d1 - Se * d2
    put = call - P + Se

    return call, put
```

Variant 3: NumExpr* (proxy for Umath implementation)



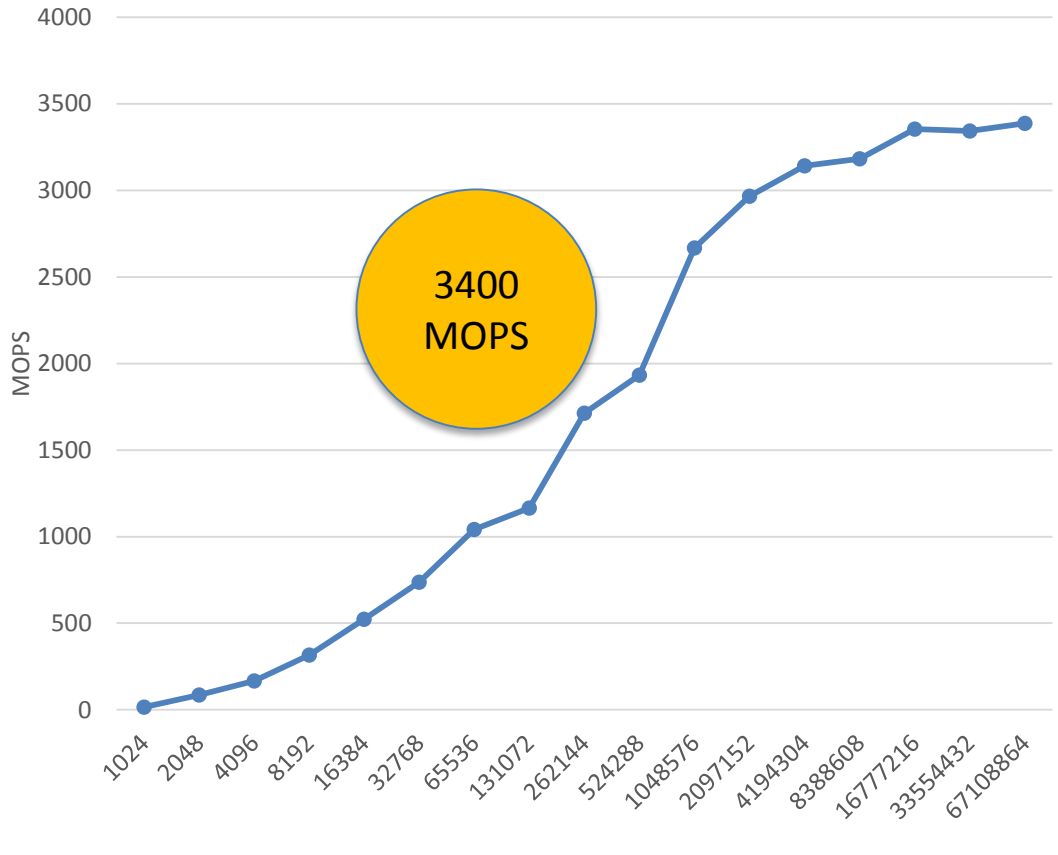

```python
import numexpr as ne

def black_scholes ( nopt, price, strike, t, rate, vol ):
      mr = -rate
      sig_sig_two = vol * vol * 2

      P = price
      S = strike
      T = t

      a = ne.evaluate("log(P / S) ")
      b = ne.evaluate("T * mr ")

      z = ne.evaluate("T * sig_sig_two ")
      c = ne.evaluate("0.25 * z ")
      y = ne.evaluate("1/sqrt(z) ")

      w1 = ne.evaluate("(a - b + c) * y ")
      w2 = ne.evaluate("(a - b - c) * y ")

      d1 = ne.evaluate("0.5 + 0.5 * erf(w1) ")
      d2 = ne.evaluate("0.5 + 0.5 * erf(w2) ")

      Se = ne.evaluate("exp(b) * S ")

      call = ne.evaluate("P * d1 - Se * d2 ")
      put = ne.evaluate("call - P + Se ")

      return call, put

ne.set_num_threads(ne.detect_number_of_cores())
base_bs_erf.run("Numexpr", black_scholes)
```

Variant 4: NumExpr* (most performant)



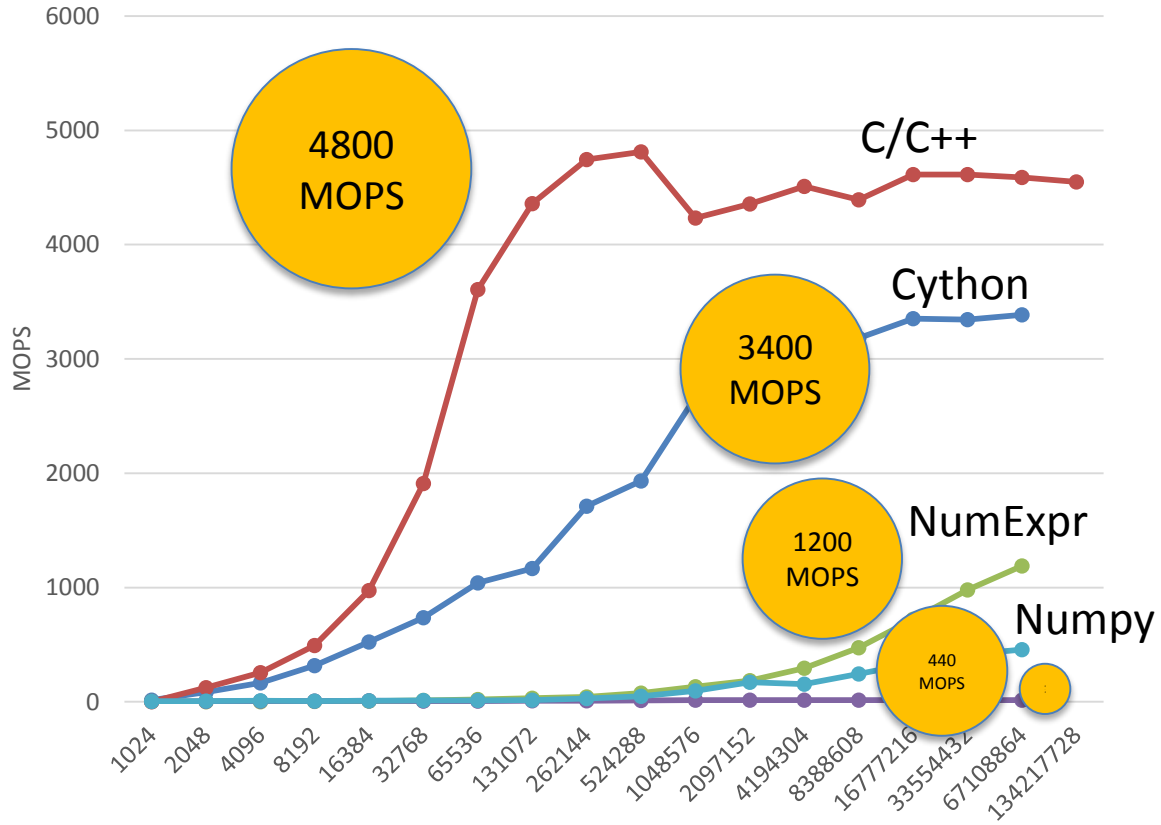

```python
import base_bs_erf
import numexpr as ne

def black_scholes ( nopt, price, strike, t, rate, vol ):
    mr = -rate
    sig_sig_two = vol * vol * 2

    P = price
    S = strike
    T = t

    call = ne.evaluate("P * (0.5 + 0.5 * erf((log(P / S) - T * mr +" +
    "0.25 * T * sig_sig_two) * 1/sqrt(T * sig_sig_two))) - S * exp(T * mr)*" +
    "(0.5 + 0.5 * erf((log(P / S) - T * mr - 0.25 * T * sig_sig_two) *" +
    "1/sqrt(T * sig_sig_two))) ")
    put = ne.evaluate("call - P + S * exp(T * mr) ")

    return call, put
```

Variant 5: Cython*




```
# In order to release GIL for a parallel loop, the code in this block cannot
# manipulate Python objects in any way.
@boundscheck(False)
@wraparound(False)
@cdivision(True)
@initializedcheck(False)
def black_scholes(int nopt,
                  double[:] price,
                  double[:] strike,
                  double[:] t,
                  double rate,
                  double vol,
                  double[:] call,
                  double[:] put):

    cdef int i
    cdef double P, S, a, b, z, c, Se, y, T
    cdef double d1, d2, w1, w2
    cdef double mr = -rate
    cdef double sig_sig_two = vol * vol * 2

    with nogil, parallel():
        for i in prange(nopt):
            P = price [i]
            S = strike [i]
            T = t [i]

            a = log(P / S)
            b = T * mr

            z = T * sig_sig_two
            c = 0.25 * z
            y = 1/sqrt(z)

            w1 = (a - b + c) * y
            w2 = (a - b - c) * y

            d1 = 0.5 + 0.5 * erf(w1)
            d2 = 0.5 + 0.5 * erf(w2)

            Se = exp(b) * S

            call [i] = P * d1 - Se * d2
            put [i] = call [i] - P + Se
```

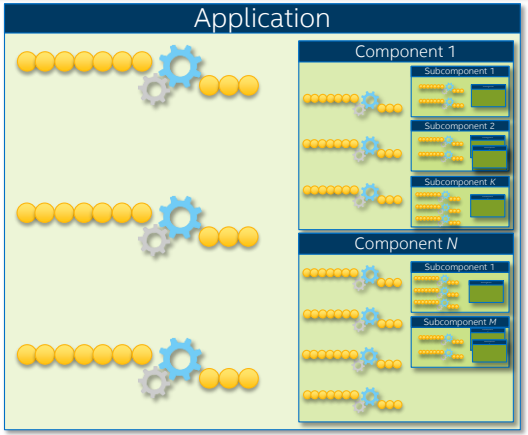
Variant 5: Native C/C++ *vs.* Python variants

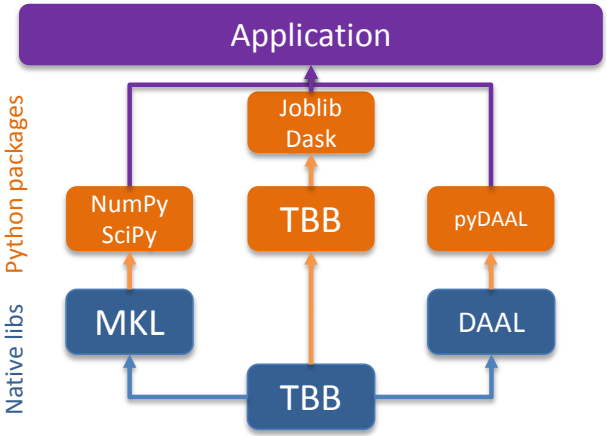


Composable parallelism

Composable Multi-Threading With Intel® TBB

- Amhdal's law suggests extracting parallelism at all levels
- If software components do not coordinate on threads use it may lead to oversubscription
- Intel TBB dynamically balances HW thread loads and effectively manages oversubscription
- Intel engineers extended Cpython* and Numba* thread pools with support of Intel® TBB



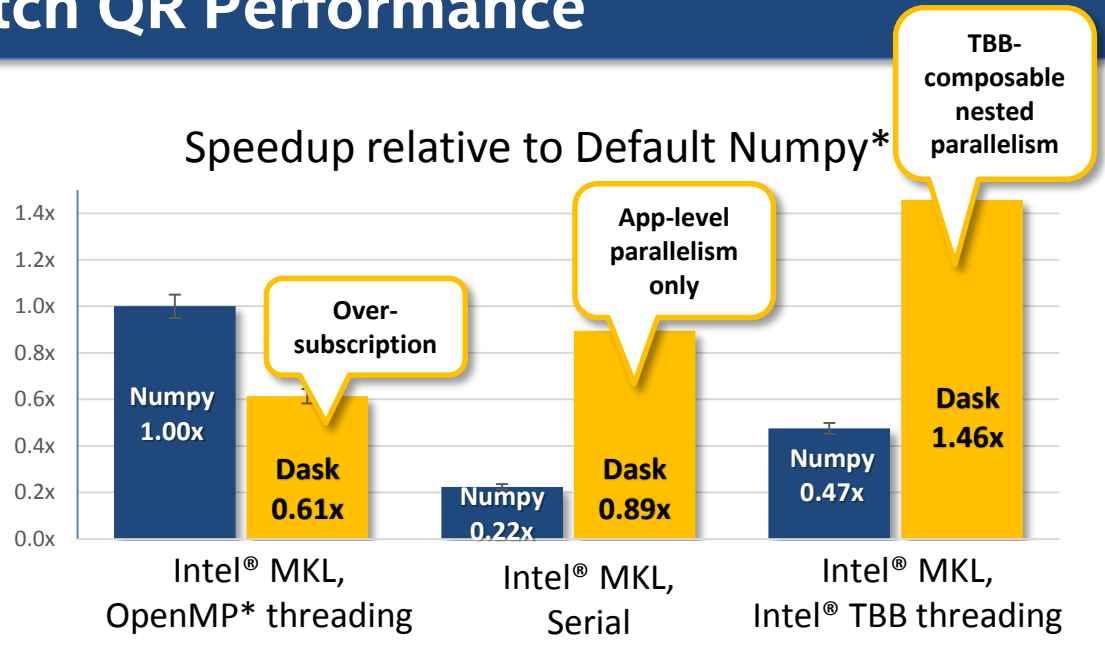



```
>python –m TBB myapp.py
```

Composable Multi-Threading Example: Batch QR Performance

```python
import time, numpy as np
x = np.random.random((100000, 2000))
t0 = time.time()
q, r = np.linalg.qr(x)
test = np.allclose(x, q.dot(r))
assert(test)
print(time.time() - t0)
```

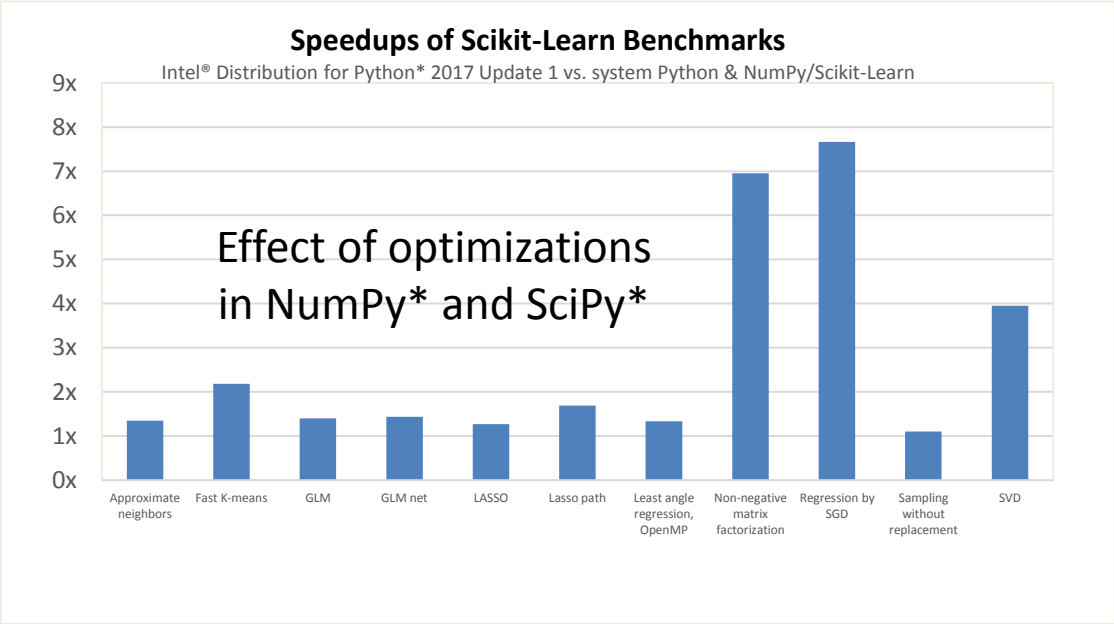
```python
import time, dask, dask.array as da
x = da.random.random((100000, 2000),
                     chunks=(10000, 2000))
t0 = time.time()
q, r = da.linalg.qr(x)
test = da.all(da.isclose(x, q.dot(r)))
assert(test.compute()) # threaded
print(time.time() - t0)
```



System info: 32x Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz, disabled HT, 64GB RAM; Intel(R) MKL 2017.0 Beta Update 1 Intel(R) 64 architecture, Intel(R) AVX2; Intel(R)TBB 4.4.4; Ubuntu 14.04.4 LTS; Dask 0.10.0; Numpy 1.11.0.

Machine Learning

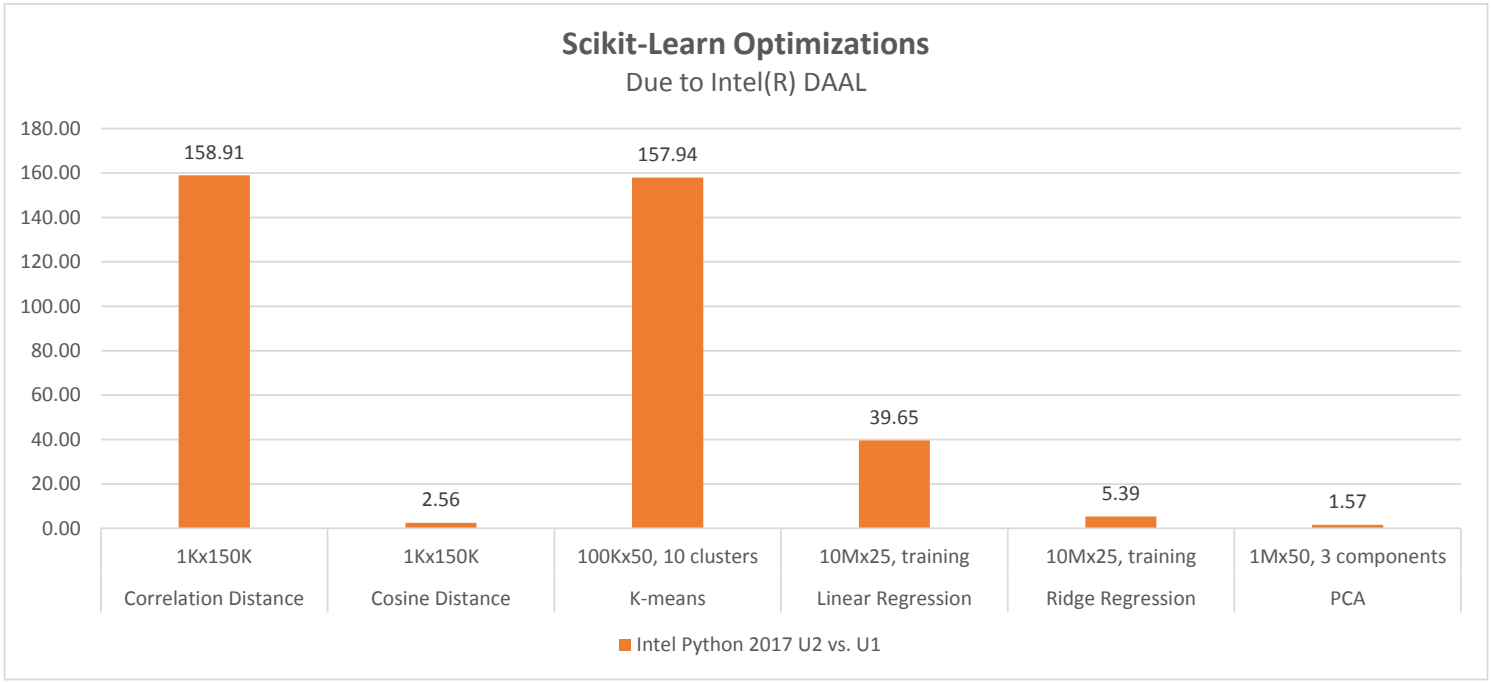
Skt-Learn* Optimizations With Intel® MKL



System info: 32x Intel® Xeon® CPU E5-2698 v3 @ 2.30GHz, disabled HT, 64GB RAM; Intel® Distribution for Python* 2017 Gold; Intel® MKL 2017.0.0; Ubuntu 14.04.4 LTS; Numpy 1.11.1; scikit-learn 0.17.1.

More Scikit-Learn* optimizations with pyDAAL

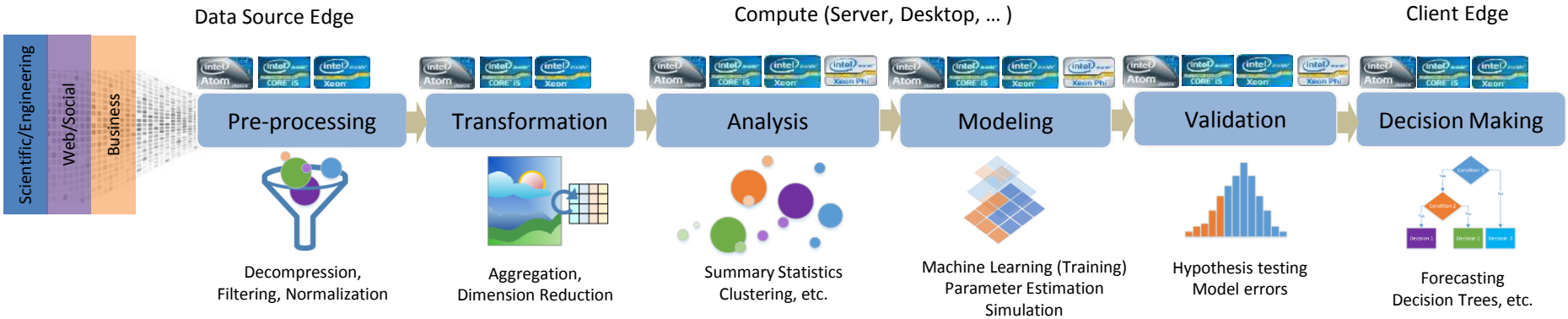
- Accelerated key Machine Learning algorithms with Intel DAAL
  - Distances, K-means, Linear & Ridge Regression, PCA
  - Up to 160x speedup on top of MKL initial optimizations



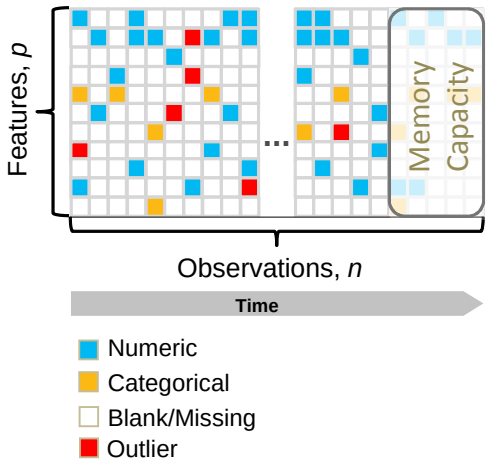
Why you may need pyDAAL in addition to Scikit-learn

Ideas Behind Intel® DAAL: Heterogeneous Analytics

- Data is different, data analytics pipeline is the same
- Data transfer between devices is costly, protocols are different
  - Need data analysis proximity to Data Source
  - Need data analysis proximity to Client
  - Data Source device ≠ Client device
  - Requires abstraction from communication protocols



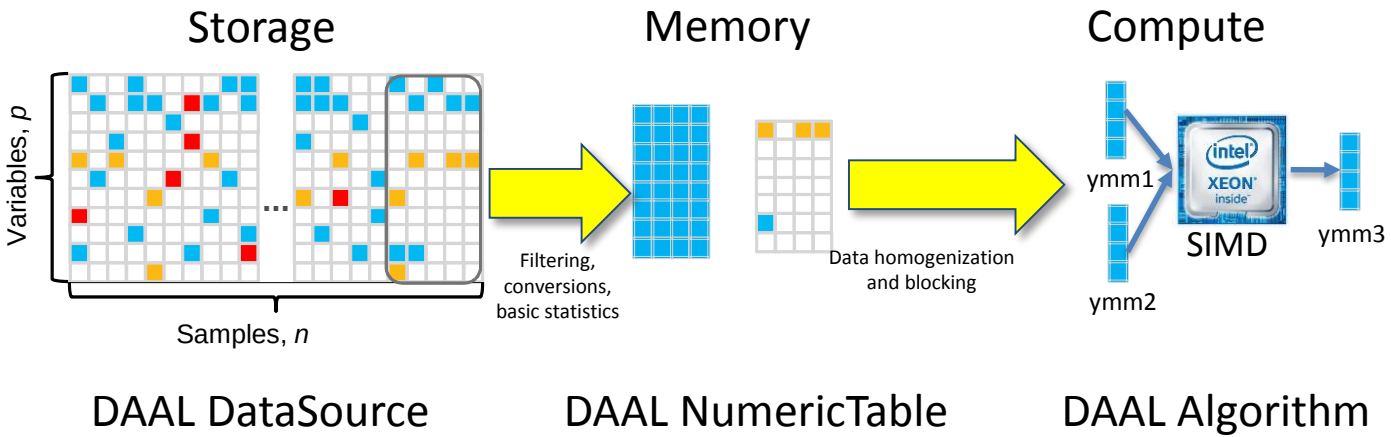
Ideas Behind Intel® DAAL: Effective Data Management, Streaming and Distributed Processing



Big Data Attributes	Computational Solution
Distributed across different devices	•Distributed processing with communication-avoiding algorithms
Huge data size not fitting into device memory	•Distributed processing •Streaming algorithms
Data coming in time	•Data buffering & asynchronous computing •Streaming algorithms
Non-homogeneous data	•Categorical→Numeric (counters, histograms, etc) •Homogeneous numeric data kernels • Conversions, Indexing, Repacking
Sparse/Missing/Noisy data	•Sparse data algorithms •Recovery methods (bootstrapping, outlier correction)

Ideas Behind Intel® DAAL: Storage & Compute

- Optimizing storage ≠ optimizing compute
  - Storage: efficient non-homogeneous data encoding for smaller footprint and faster retrieval
  - Compute: efficient memory layout, homogeneous data, contiguous access
  - Easier manageable for traditional HPC, much more challenging for Big Data



Ideas Behind Intel® DAAL: Languages & Platforms

DAAL has multiple programming language bindings

- C++ – ultimate performance for real-time analytics with DAAL
- Java*/Scala* – easy integration with Big Data platforms (Hadoop*, Spark*, etc)
- Python* – advanced analytics for data scientist

Performance profiling with Intel® VTune™ Amplifier

Profiling Python* code with Intel® VTune™ Amplifier

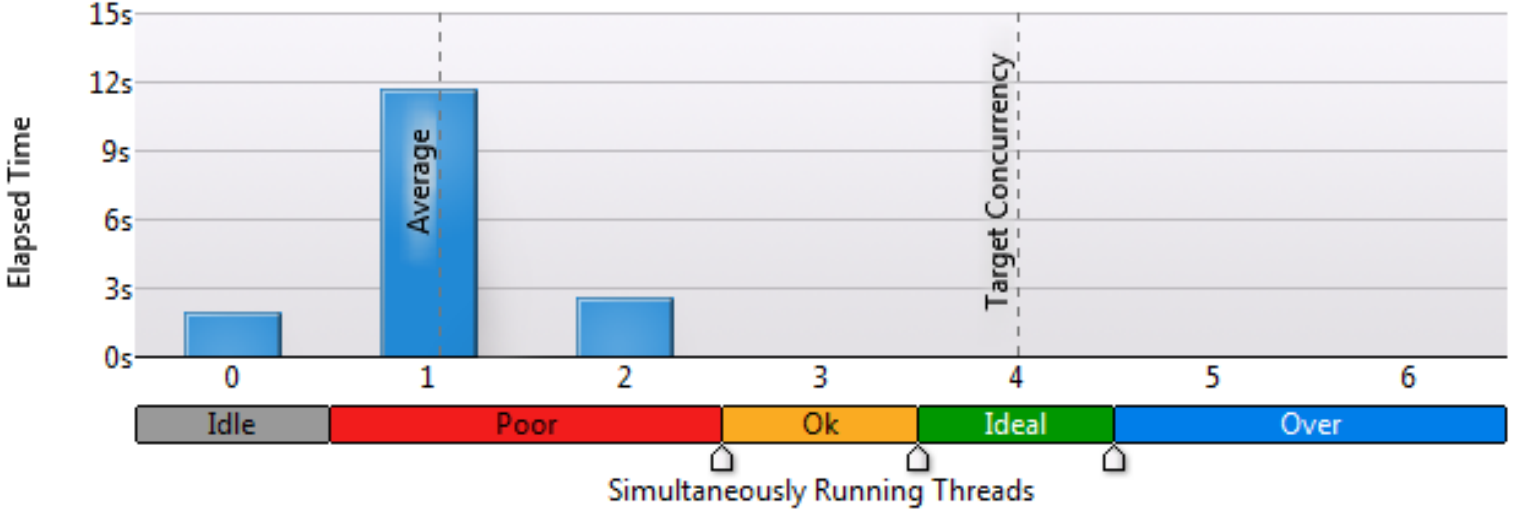
- Right tool for high performance application profiling at all levels
  - Function-level and line-level hotspot analysis, down to disassembly
  - Call stack analysis
  - Low overhead
  - Mixed-language, multi-threaded application analysis
  - Advanced hardware event analysis for native codes (Cython, C++, Fortran) for cache misses, branch misprediction, etc.

Feature	cProfile	Line_profiler	Intel® VTune™ Amplifier
Profiling technology	Event	Instrumentation	Sampling, hardware events
Analysis granularity	Function-level	Line-level	Line-level, call stack, time windows, hardware events
Intrusiveness	Medium (1.3-5x)	High (4-10x)	Low (1.05-1.3x)
Mixed language programs	Python	Python	Python, Cython, C++, Fortran

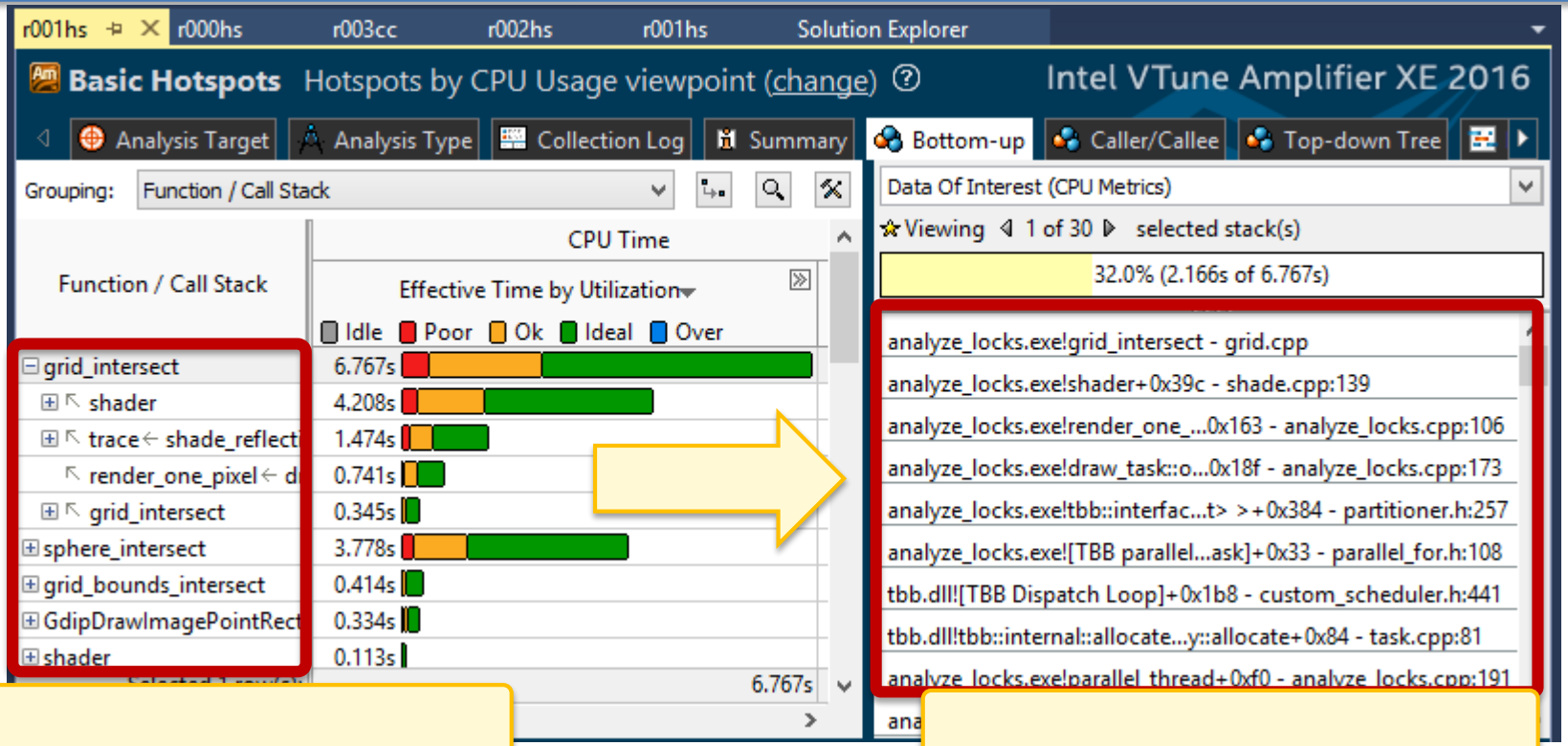
Intel® VTune™ Amplifier XE
1. Get a quick snapshot

Thread Concurrency Histogram

This histogram represents a breakdown of the Elapsed Time. It visualizes the percentage of the wall time the specific number of threads were considered running if they are either actually running on a CPU or are in the runnable state in the OS scheduler. Essentially, Thread Concurrer that were not waiting. Thread Concurrency may be higher than CPU usage if threads are in the runnable state and not consuming CPU time.



Intel® VTune™ Amplifier XE
2. Identify Hotspots



Hottest Functions

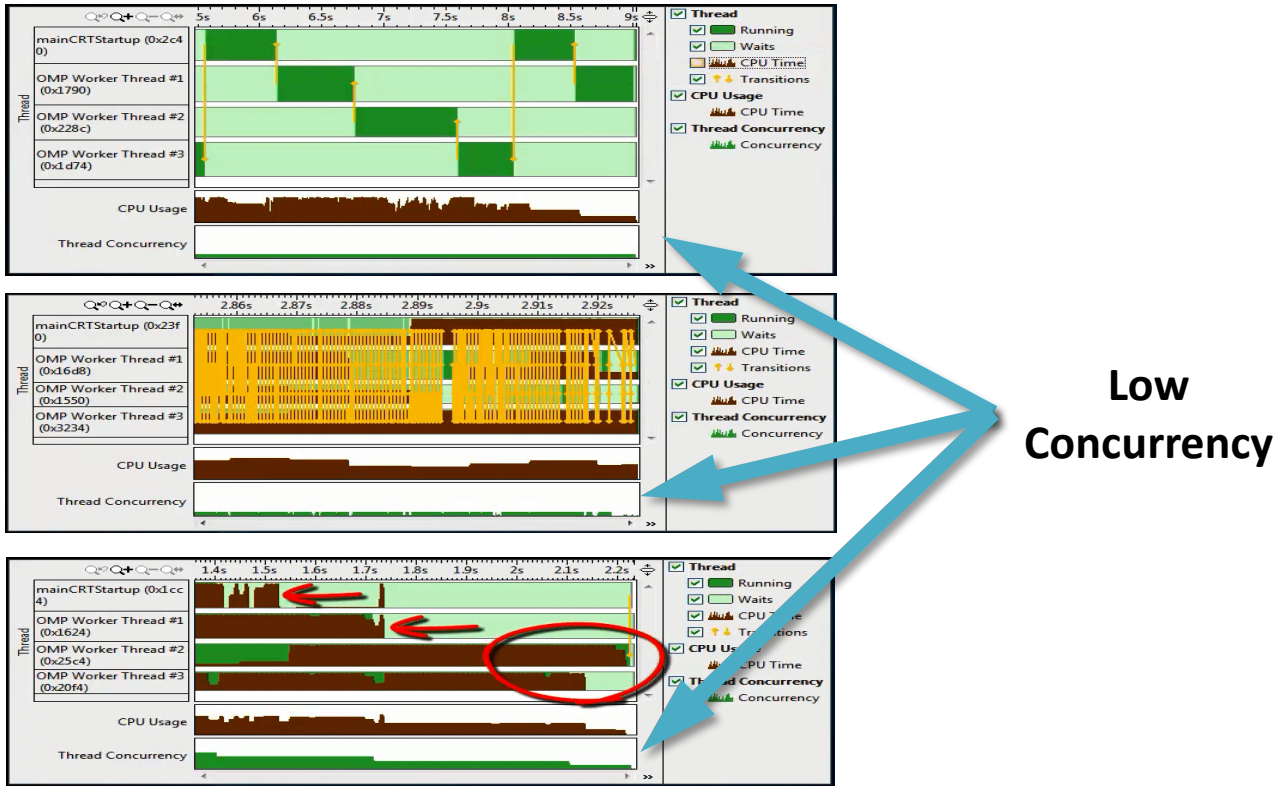
Hottest Call Stack

Intel® VTune™ Amplifier XE
3. Look for common patterns

Coarse Grain Locks

High Lock Contention

Load Imbalance



Low Concurrency

Intel® VTune™ Amplifier XE Navigation through your code

Adjust Data Grouping

- Function - Call Stack
- Module - Function - Call Stack
- Source File - Function - Call Stack
- Thread - Function - Call Stack

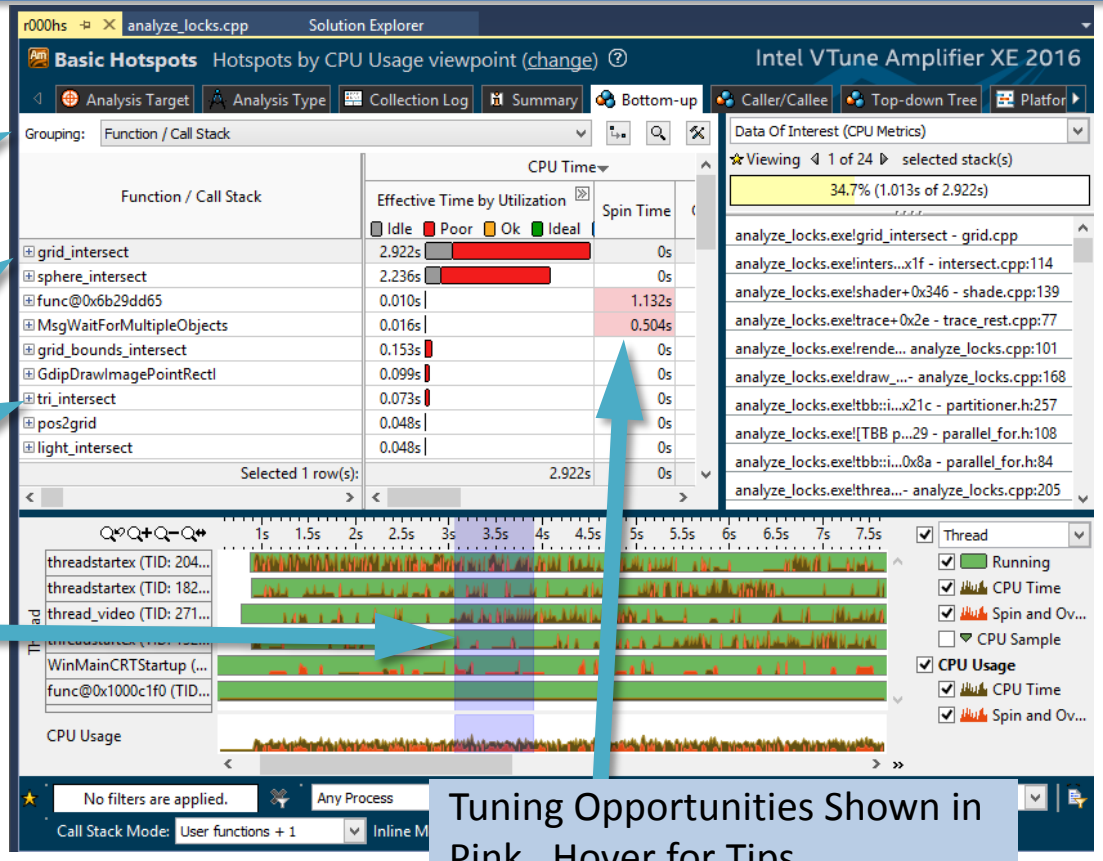
... (Partial list shown)

Double Click Function to View Source

Click [+] for Call Stack

Filter by Timeline Selection (or by Grid Selection)

- Zoom In And Filter On Selection
- Filter In by Selection
- Remove All Filters



Tuning Opportunities Shown in Pink. Hover for Tips

Summary And Call To Action

- Intel created the Python* distribution for out-of-the-box performance and scalability on Intel® Architecture
  - With minimum to no code modification Python aims to scale
- Multiple technologies applied to unlock parallelism at all levels
  - Numerical libraries, libraries for parallelism, Python code compilation/JITing, profiling
  - Enhancing mature Python packages and bringing new technologies, e.g. pyDAAL, TBB
- With multiple choices available Python developer needs to be conscious what will scale best
  - Intel® VTune™ Amplifier helps making conscious decisions

Intel Distribution for Python is free!

https://software.intel.com/en-us/intel-distribution-for-python

Commercial support included for Intel® Parallel Studio XE customers!

Easy to install with Anaconda* https://anaconda.org/intel/